```
/* Yiqing Cao(Grace)
 *
 * Program name: OS4
 * Purpose: A simulation to implement the first-fit memory allocation scheme.
 * Input  : Prompts for the name of file that each line containes op, PID and size for a process.
 * Output : An aligned memory table display the PID, beginning address, ending address and size
 *          of any process.
 *          Number of processes (and their PIDs) that did not fit in memory.
 *          Total amount of memory in use at the end of the simulation.
 *          Total amount of memory remaining.
 *          Number of contiguous spaces (memory blocks) remaining.
 *          Average size of remaining memory blocks.
 */

#include<iostream>
#include<fstream>
#include<stdlib.h>
#include<time.h>
#include<queue>

using namespace std;

// global constant
const int MAINMEMORYSIZE = 1048576; // 1*1024*1024

// class of all the functions and variables needed for simulation
class Simulation{
public:
    struct Process{
        char op;
        int pid;
        int size;
        int begin;
        int end;
    };
    vector<Process> incoming; // vector containes all the processes from the input file
    vector<Process> JobLeft; // vector stores processes that did not fit in memory
    int NumOfD;
    int count;
    int NumOfJobLeft;
    int LatestEndPosition;
    // function that read from the input file and store them into vector
    void ReadInFile();
    // function that given a pid and a vector then find previous index
    int FindPid(int, vector<Process>&);
    // function that do the simulation
    void ProcessJob();
    // function that print the output
    void Print();
};
```

```
int main (){
    Simulation FirstFit;
    FirstFit.ReadInFile();
    FirstFit.ProcessJob();
    FirstFit.Print();
    return 0;
}


void Simulation::ReadInFile(){
    string FileName;
    cout << "Please enter the file name: " << endl;
    cin >> FileName;

    ifstream fin;
    fin.open(FileName.c_str());
    if(fin.is_open()){ // check if file is open
        Process NewProcess;
        while(fin){
            fin >> NewProcess.op;
            if(NewProcess.op != 'Q'){
                if(NewProcess.op != 'D'){
                    fin >> NewProcess.pid >> NewProcess.size;
                }
                else{
                    fin >> NewProcess.pid;
                    NewProcess.size = 0;
                    NewProcess.begin = 0;
                    NewProcess.end = 0;
                }
            }
            else{
                NewProcess.pid = -1;
                NewProcess.size = 0;
                NewProcess.begin = 0;
                NewProcess.end = 0;
            }
            incoming.push_back(NewProcess); // push process info into vector
        }
    }
    else{
        cout << "No such file exists!" << endl;
    }
}

int Simulation::FindPid(int Pid, vector<Process>& incoming){
    bool found = 0; // boolean to indicate whether found or not
    int index = -1;
```

```
    while(!found){
        index++;
        if(Pid == incoming[index].pid){
            found = 1;
        }
    }
    if(found == 1){
        return index;
    }
    else{
        return -1;
    }
}


void Simulation::ProcessJob(){
    int index;
    NumOfD = 0;
    count = 0;
    NumOfJobLeft = 0;
    LatestEndPosition = 0;

    int i = 0;
    while(incoming[i].op != 'Q'){ // run through all the process until reach the end
        bool insert = 0; // boolean that check whether the process is allocated or not
        if(incoming[i].op == 'A'){ // allocate the process into memory
            if(incoming[i].size <= MAINMEMORYSIZE){
                int k = 0;
                while(insert == 0 && k < incoming.size()){
                    if(incoming[k].pid == 0 && incoming[k].size >= incoming[i].size){ // allocate into free block
                        incoming[i].begin = incoming[k].begin;
                        incoming[i].end = incoming[i].begin + incoming[i].size - 1;
                        incoming[k].begin += incoming[i].size;
                        incoming[k].size -= incoming[i].size;
                        insert = 1;
                        incoming.insert(incoming.begin()+k, incoming[i]);
                        incoming.erase(incoming.begin()+i+1);
                    }
                    k++;
                }
                if(insert == 0){ // allocate to end if there are enough spaces
                    if(LatestEndPosition + incoming[i].size <= MAINMEMORYSIZE){
                        incoming[i].begin = LatestEndPosition;
                        LatestEndPosition += incoming[i].size;
                        incoming[i].end = incoming[i].begin + incoming[i].size - 1;
                    }
                    else{
                        JobLeft.push_back(incoming[i]);
                        NumOfJobLeft++;
                        incoming.erase(incoming.begin() + i);
```

```
                        i--;
                    }
                }
            }
            else{
                JobLeft.push_back(incoming[i]);
                NumOfJobLeft++;
            }
        }
        else if(incoming[i].op == 'D'){ // deallocate the process
            NumOfD++;
            index = FindPid(incoming[i].pid, incoming);
            if(index != -1){
                incoming[index].pid = 0;
                incoming.erase(incoming.begin() + i);
                i--;
            }
            for(int k = 0; k < incoming.size() - NumOfD; k++){
                //check if there are contiguous spaces then coalescing them
                if(incoming[k].pid == 0 && incoming[k+1].pid == 0){
                        incoming[k].end = incoming[k+1].end;
                        incoming[k].size += incoming[k+1].size;
                        incoming[k+1].size = 0;
                        incoming.erase(incoming.begin() + (k+1));
                        k--;
                        i--;
                        count++;
                }
                // check whether a free block still have spaces or not
                if(incoming[k].pid == 0 && incoming[k].size == 0){
                    incoming.erase(incoming.begin() + (k));
                    i--;
                    count++;
                }
            }
        }
    i++;
    }
}


void Simulation::Print(){
    cout << "\tMemory Table\n";
    cout << "PID\tBegin\tEnd\tSize\n";
    for(int i = 0; i <= incoming.size() - count; i++){
        if(incoming[i].pid == 0){
            cout << "Free\t" << incoming[i].begin << "\t" << incoming[i].end << "\t" << incoming[i].size << "\n";
        }
        else{
            cout << incoming[i].pid << "\t" << incoming[i].begin << "\t" << incoming[i].end << "\t" << incoming[i].size << "\n";
```

```
        }
    }
    cout << "Free\t" << LatestEndPosition << "\t" << MAINMEMORYSIZE << "\t" << MAINMEMORYSIZE - LatestEndPosition + 1 << "\n";

    int TotalFree = MAINMEMORYSIZE - LatestEndPosition + 1;
    int NumOfFree = 1;
    for(int i = 0; i <= incoming.size() - count; i++){
        if(incoming[i].pid == 0){
            NumOfFree++;
            TotalFree += incoming[i].size;
        }
    }
    int TotalMemoryUsed = MAINMEMORYSIZE - TotalFree + 1;
    float AveSizeOfFree = TotalFree / float(NumOfFree);

    cout << "Number of processes that did not fit in memory: " << NumOfJobLeft << "\n";
    cout << "PIDs: ";
    for(int i = 0; i < JobLeft.size(); i++){
        cout << JobLeft[i].pid << "\t";
    }
    cout << "\n";
    cout << "Total amount of memory in use at the end of the simulation: " << TotalMemoryUsed << "\n";
    cout << "Total amount of memory remaining: " << TotalFree << "\n";
    cout << "Number of contiguous spaces (memory blocks) remaining: " << NumOfFree << "\n";
    cout << "Average size of remaining memory blocks: " << AveSizeOfFree << "\n";
}
```